The Basics of UNIX/Linux

11-3. Arrays and Pointer. Part 3

Instructor: Joonho Kwon
jhkwon@pusan.ac.kr
Data Science Lab @ PNU

datalab
data science laboratory

Lecture Outline

- **Pointers and Arrays**
- Function Pointers

Pointers and Arrays

- A pointer can point to an array element
  - You can use array indexing notation on pointers
    - `ptr[i]` is `*(ptr+i)` with pointer arithmetic – get the data `i` elements forward from `ptr`
  - An array name will provide the beginning address of the array
    - *Like* a pointer to the first element of array, but can't change

```
int a[] = {10, 20, 30, 40, 50};
int* p1 = &a[3];  // refers to a's 4th element
int* p2 = &a[0];  // refers to a's 1st element
int* p3 = a;      // refers to a's 1st element

*p1 = 100;
*p2 = 200;
p1[1] = 300;
p2[1] = 400;
p3[2] = 500;      // final: 200, 400, 500, 100, 300
```

Array Parameters

- Array parameters are *actually* passed as pointers to the first array element
  - The `[]` syntax for parameter types is just for convenience

This code:

```
void f(int a[]);

int main( ... ) {
  int a[5];
  ...
  f(a);
  return 0;
}

void f(int a[]) {
```

Equivalent to:

```
void f(int* a);

int main( ... ) {
  int a[5];
  ...
  f( &a[0] );
  return 0;
}

void f(int* a) {
```

Pointers vs. Array (1/3)

	Arrays	Pointers
1D array of 5 int	Int x[5]	Int *xPtr
2D array of 6 int 2x3 matrix	Int y[2][3];	Int **yPtr;
2D array of 4 int 2x2 matrix	int(*z)[2]={{1,2},{2,1}};	Int **zPtr;
1D array of 5 char string	char c[] = “mike”;	char *cPtr;

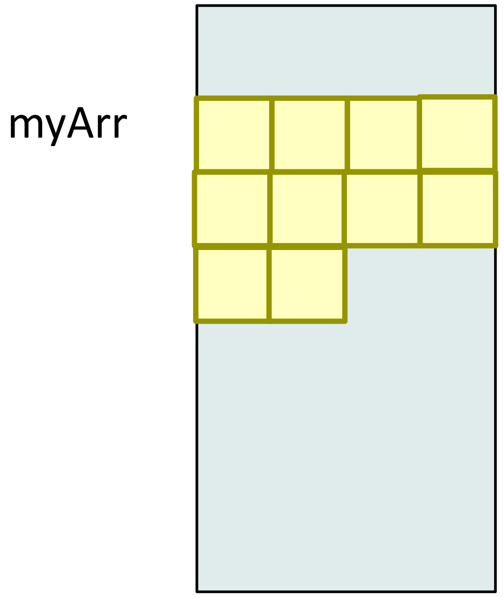
Space has been allocated in memory for the arrays

Space has been allocated in memory only for the pointers variables, **NOT** for the arrays they will point to. The DIMENSIONS of the arrays are UNKNOWN

Pointers vs. Array (2/3)

- Arrays
  - represent actual memory **allocated** space

```
char myArr[10];
```



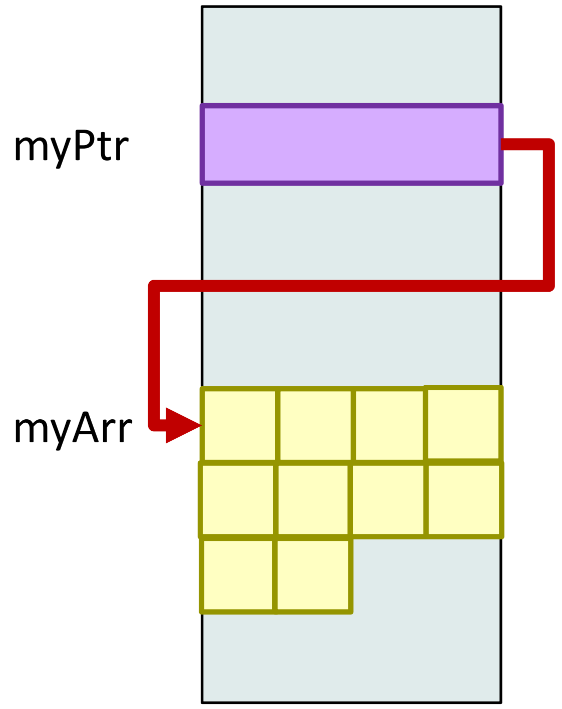
Pointers vs. Array (3/3)

- Pointers **point** to a place in memory

```
char myArr[10];
```

```
char *myPtr;
```

```
myPtr = myArr;
```



Lecture Outline

- Pointers and Arrays
- **Function Pointers**

Function Pointers

$$\begin{tikzcd}\text{jmp foo} \arrow[r, bend left, "\text{address}"] & \text{PC}\end{tikzcd}$$

- Based on what you know about assembly, what is a function name, really?
  label → address is
  - Can use pointers that store addresses of functions!
- Generic format:

function pointer → int (*fp)(int) = foo;
function prototype → int foo(int);
function prototype → int *fp (int)

pointer!

```
returnType (* name)(type1, …, typeN)
```

  - Looks like a function prototype with extra `*` in front of name
  - Why are parentheses around `(* name)` needed?

    to differentiate it from a function prototype

- Using the function:

dereference

```
(*name)(arg1, …, argN)
```

  - Calls the pointed-to function with the given arguments and return the return value

Function Pointer Declaration

- One easy way for declaration:
  - write your normal function declaration like:
    - Int myFunc(int a, int b)
    - this is a function with two int arguments and returns int value.
  - wrap function name with the pointer syntax:
    - Int (*myFunc) (int a, int b)
  - change the function name to a pointer name:
    - Int (*comparer) (int a, int b)
    - it points to a function with two integer arguments, where that function returns an integer value

Function Pointers –Similarity and Differences

- Differences with data pointers:
  - they point to code instead of data
  - we don’t allocate or deallocate memory for this type of pointers
  - you can use, either function name or &function name to assign its address to a function pointer.
- Similarity to data pointers:
  - we can define array of function pointers, where each elements refer to one function.
  - a function pointer can be passed as an argument to a function or be return from a function.

Function Pointer Example

- `map()` performs operation on each element of an array

map.c

```c
#define LEN 4

int negate(int num) {return -num;}
int square(int num) {return num*num;}

// perform operation pointed to on each array element
void map(int a[], int len, int (* op)(int n)) {
  for (int i = 0; i < len; i++) {
    a[i] = (*op)(a[i]);  // dereference function pointer
  }
}

int main(int argc, char** argv) {
  int arr[LEN] = {-1, 0, 1, 2};
  int (* op)(int n);   // function pointer called 'op'
  op = square;      // function name returns addr (like array)
  map(arr, LEN, op);
  ...
```

funcptr parameter

funcptr dereference

funcptr definition

funcptr assignment

Q&A